Greedy Technique

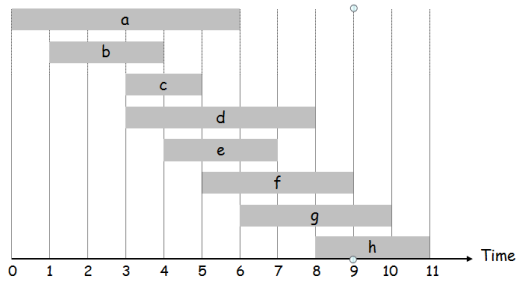
- Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:
  - *feasible*
  - *locally optimal*
  - *irrevocable*
- For some problems, yields an optimal solution for every instance.
- For some problems where it does not yield an optimal solution it can be useful for fast approximations

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes
- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Interval Scheduling

- Interval scheduling.
  - Job j starts at sj and finishes at fj.
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.
  - [Earliest start time] Consider jobs in ascending order of start time sj.
  - [Earliest finish time] Consider jobs in ascending order of finish time fj.
  - [Shortest interval] Consider jobs in ascending order of interval length fj - sj.
  - [Fewest conflicts] For each job, count the number of conflicting jobs cj. Schedule in ascending order of conflicts cj.

Interval Scheduling: Greedy Algorithms

- Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

Interval Scheduling: Greedy Algorithm

- Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken. (Earliest finish time first)

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
                    jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

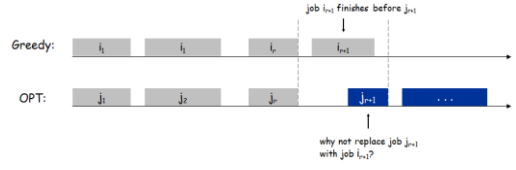
- Implementation. O(n log n).
  - Remember job j* that was added last to A.
  - Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling – Stay Ahead proofs Proof of correctness

- See handout on PolyLearn for a proof by induction

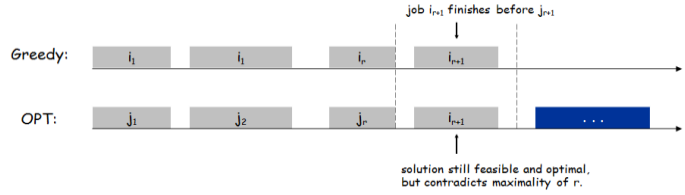
Interval Scheduling: Theorem. Greedy (E.F.T.F) algorithm is optimal

- Pf. (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution which has the largest set initial set of jobs that are the same as those in the greedy solution, m>k. The figure below shows this assuming the first r < k jobs are the same.
  - Since the i_{r+1} job has the earliest finish time compatible with $i_r = j_r$ it must finish before j_{r+1}. Replace j_{r+1} with i_{r+1} and the remaining jobs in the optimal solution will still be compatible with $i_1 \ldots i_{r+1}$.



Theorem. Greedy algorithm is optimal.

- Pf. (continued)
  - This forms a solution of m compatible jobs that agree with the greedy solution for the first r+1 jobs .
  - This contradicts the assumption that the longest initial agreement between an optimal solution with the greedy was r<k jobs.
  - Since there is an optimal solution with r=k we have the situation below. But it is impossible that there are remaining jobs compatible with job k since greedy would have chose one.



Total Weighted Time to Completion Problem

Total Weighted Time to Completion Problem: Given a set of jobs: job_1, job_2, …, job_n characterized by their lengths: l_1, l_2, … , l_n and their weights: w_1, w_2, … , w_n

- Output a schedule that minimizes the total weighted time to completion.
  - Schedule: An ordering of the jobs to be serviced by the resource
  - Completion time: C_i, of job l_i is the sum of the l_j's up to and including l_i
- Total Weighted Time to Completion = $\sum_{i=1 \text{ to } n} w_i C_i$

Question?

- Given 3 jobs job_1, job_2, job_3 where the respective lengths are 1, 2, 3. What are the Completion times of the three jobs if done in the order given?
  a) 1, 2, 3
  b) 3, 5, 6
  c) 1, 3, 6
  d) 1, 4, 6
- If the weights are respectively 3, 2, 1. What is the Total Weighted Time to Completion for the ordering 1, 2, 3 ?

Note: If there are n jobs then there are n! orderings.

Total Weighted Time to Completion Problem (TWTC)

Finding a Greedy algorithm that solves this problem:
Focus on process

- Goal: Find a sequence of a set of jobs that **minimizes** the total weighted time to completion.
- There may be many candidate greedy algorithms!
- How do we find a greedy algorithm that works.
  - Look at special cases
  - Generate candidate greedy algorithms
  - Narrow down to single (if possible) candidate, usually by looking at simple cases to eliminate candidates, then prove correctness. (It may be that no greedy algorithm works.)

Discovery of candidates

- Explore special cases:
  - Equal lengths
  - Equal weights
- What order do you think you should schedule jobs?
  a) Larger weights first; shorter lengths first
  b) Smaller weights first; shorter lengths first
  c) Larger weights first; longer lengths first
  d) Smaller weights first; longer lengths first
- Hint: Look at small examples to test your ideas

What are reasonable candidates?

- How can we combine the information we have into a greedy algorithm
- Two simple options, order the jobs by either:
  - Difference: $w_i - l_i$ Ratio: w_i / l_i
- How to rule one of these out?
  - Find, if possible, an example where the two different algorithms give two different answers.
  - Clearly the larger solution for the example is not optimal.
  - This rules out that algorithm but does not prove anything about the algorithm that gives the smaller solution.

TWTC example problem

Consider 2 jobs $l_1 = 1, w_1 = 5$ $l_2 = 4, w_2 = 12$

- What is the weighted total time to completion given by the two algorithms?

2 jobs $l_1 = 1, w_1 = 5$ $l_2 = 4, w_2 = 12$

- What is the weighted total time to completion given by the two algorithms?

	$W_i - L_i$	W_i / L_i
Job_1	4	5
Job_2	8	3
Ordering	Job_2 , Job_1	Job_1 , Job_2
TWTC	48+25 = 73	5+60 =65

- $W_i - L_i$ does not always yield the optimal (smallest) TWTC.

Proving correctness: exchange argument

Plan: Assume the greedy algorithm does not produce the optimal solution.

- Assume the w/l ratios are all distinct so there is only one possible ordering given by the algorithm.
- Assume that the jobs are labeled according to the decreasing w/l ratio, e.g. w_1/l_1 is the largest ratio, w_n/l_n is smallest
- Assume that there is an ordering $\sum^*$ that is the optimal solution but is different from $\sum$ the ordering given by the algorithm

Now show that $\sum^*$ is not the optimal solution.

This contradicts our assumption that $\sum^*$ is optimal and different from $\sum$. Hence the optimal solution must be $\sum$.

Proving correctness: exchange argument details

the jobs are labeled according to the decreasing ratios

If $\sum^*$ is not $\sum$, it must have two consecutive jobs where i > j (i has the smaller ratio) but i comes before j in $\sum^*$. Why must there be two consecutive jobs out of order ?

Order of jobs in $\sum^*$

Jobs before i	Job i	Job j	Jobs after j

If switch jobs i and j, what happens to the completion times of jobs?

Proving correctness: exchange argument details

Jobs before i	Job i	Job j	Jobs after j

If switch jobs i and j, what happens to the completion times of jobs?

	$k \neq i, j$	i	j
–	unaffected	up	down

- Now what does switching i and j do to the total cost?
  - Since i > j we know that $w_I / l_I < w_J / l_J$ so
  - $w_I * l_J < w_J * l_I$ means that $w_J * l_I - w_I * l_J > 0$
  - Claim: the ordering after making the switch has a lower total weighted time to completion! CONTRADICTION that $\sum^*$ was the minimum total weighted time to completion

details

- Before

Jobs before i	Job i	Job j	Jobs after j

C

- After

Jobs before i	Job j	Job i	Jobs after j

C

- Weighted Completion time:

After	job_I	$w_I * (C + l_J + l_I)$	job_J	$w_J * (C + l_J)$
Before	job_I	$w_I * (C + l_I)$	job_J	$w_J * (C + l_I + l_J)$

Before – After = $w_J * l_I - w_I * l_J$ but this is >0

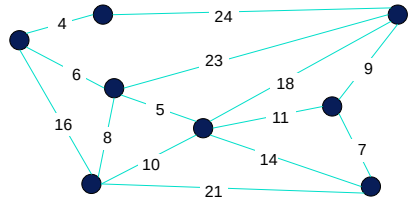
$\Rightarrow$ After has a smaller TWTC

Contradicts "Before" being optimal $\Rightarrow$ ordering by decreasing weight to length ratio gives an optimal ordering

Minimum Spanning Tree Problem

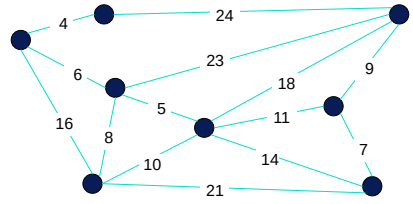
- *Spanning tree* of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices
- *Minimum spanning tree, MST,* of a weighted, connected graph *G*: a spanning tree of *G* of minimum total weight (sum of the edge weights in the tree)
- Minimum Spanning Tree Problem: For a weighted graph, find the MST

Minimal Spanning Tree Example

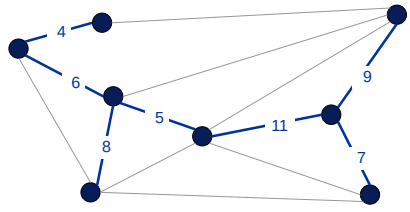


G = (V, E)

Minimal Spanning Tree Example



G = (V, E)



T, $\Sigma_{e \in T} c_e = 50$

3 Greedy Approaches – these all work!

- Build a spanning tree by
  - Successively adding the lightest edge that does not create a cycle (Kruskal's)
  - Start with a root node and grow the tree outward without creating a cycle (Prim's)
  - Start with the full graph and delete edges in order of decreasing cost as long as we do not disconnect the tree (Reverse-delete?)

Prim's MST algorithm

- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees T_1, T_2, …, T_n
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)
  - Key Question – how do we do this efficiently
- Stop when all vertices are included

Cut property – proving MST algorithms correct

Assume all the edge costs are distinct. Let

- S be non-empty proper subset of V
- e = (v,w) the minimal cost edge between S and V-S
- then every MST contains e

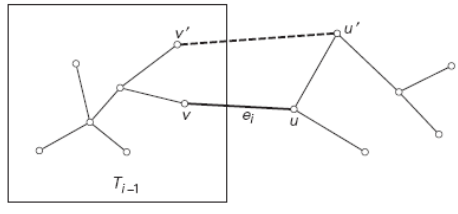


FIGURE 9.4 Correctness proof of Prim's algorithm.

Cut property – proving MST algorithms correct

Assume all the edge costs are distinct. Let

- S be non-empty proper subset of V
- e = (v,w) the minimal cost edge between S and V-S
- then every MST contains e

Proof: (sketch – fill in the details)

- Let T be a spanning tree that does not contain e
- Use an exchange argument
- Find an e' so that exchanging e for e' reduces the cost
  - » There must be a path from v to w in T
  - » Follow the path to find an edge that goes from S to S-V
  - » Let that be e', exchange to get T'
  - » Finally show: T' is a tree (acyclic) and has lower cost

Correctness: Prim's Algorithm

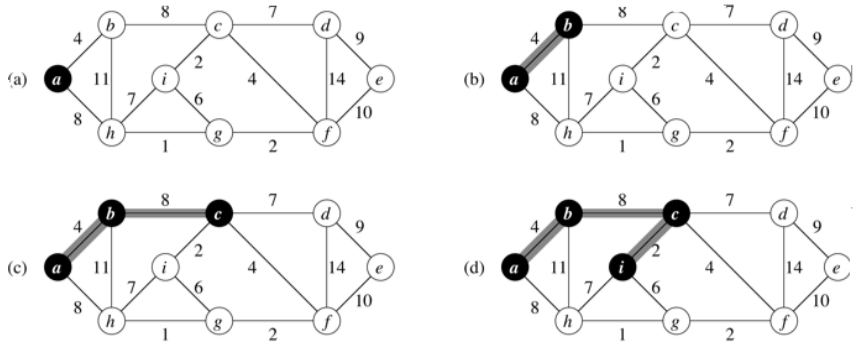
Correctness : Use induction to show that all the T i are subtrees of the MST (apply the cut property)

- Base case: T_1 = {start vertex} . Clearly a subtree of a Minimal Spanning of T
- Inductive case: T_k a subtree of an MST $\rightarrow$ T_{k+1} a subtree of an MST
- Proof of inductive case: Apply the cut property to show that the edge chosen by Prim must be in any MST.
  - Let S be the set of vertices in T_k ; V-T_k be will be the vertices not in T_k
  - By the cut property the smallest edge connecting T_k and V-T_k must be in the MST
  - But this is the edge chosen by Prim's Algorithm to add to T_k to make T_{k+1}
  - This proves the inductive set – so all the subtrees created by Prim are subtrees of the MST
  - Hence all the edges in T_{k+1} are in the MST for the graph

Analysis: Prim's Algorithm

- Analysis: Using a heap implementation of a **priority queue** the algorithm will run:
  - O(|E|) since each edge will eventually be placed in the queue
  - |V| * extractMin and |E| * changePriority operations
  - A priority queue can be implemented using a heap to so that both extractMin and changePriority are O(log |V|)
  - Therefore the overall running time is |E| * (log |V|)
- Why is it not proper to use the priority queue operations as the basic operations?

Example of Prim's Algorithm in action



Prim's Algorithm: More details

```
Initialize prev(v), dist(v) ∀ v∈V;  T= {start vertex};  U = V-T; dist(start vertex) = 0
while(T is missing a vertex)
  pick the vertex, v1, in U with the shortest edge, dist(), to the group of vertices in
     the spanning tree add v1 to T
  /* this loop looks through every neighbor of v and checks to see if that
   * neighbor could reach the minimum spanning tree more cheaply through v1 */
  for each edge of  v1 (v1, v2)
      if(length(v1, v2) < dist[v2])     // this is "relaxation"
         dist[v2] = length(v1, v2)
         prev[v2] = v1
      end if
   end for
end while
```


Single Source Shortest Paths Problem

Single Source Shortest Paths Problem: Given a weighted connected graph G, find shortest (sum of the weights of the edges) paths from source vertex *s* to each of the other vertices

- Variants:
  - Single destination shortest paths problem
  - Single pair shortest paths problem
  - All pairs shortest paths problem
- Issues
  - Negative edge weights
  - Cycles

Three Shortest Path Algorithms
Depend on idea of relaxation

- Dijkstra
  - Edge weights must be non-negative
  - Relaxes each edge exactly once. O ((log |V|)*|E|)
- Bellman-Ford
  - Works with negative edge weights (Of course not cycles!)
  - Relaxes edges multiple times O (|V|* |E|)
- DAG - assumes no cycles!
  - Uses topological sort of vertices
  - Guarantees O(|E|+|V|) performance
  - Works with negative edge weights

Relaxation:

- For each vertex, v, maintain an upper bound on the length (stored in dist(v)) of the shortest path from the source vertex to that vertex and the previous vertex on that shortest path
- Test to see if some new path is shorter than the current upper bound. If so then reduce the upper bound and update the previous vertex.

```
Relax(e=(u,v), w)
        if dist(v)>dist(u)+w(u,v)
                dist(v) = dist(u)+w(u,v)
                prev(v) = u
```

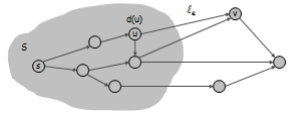
Dijkstra's Algorithm

- Dijkstra's algorithm.
  - Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  - Initialize S = { s }, d(s) = 0.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



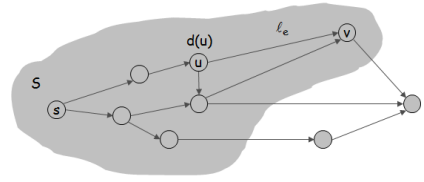
Dijkstra's Algorithm

- Dijkstra's algorithm.
  - Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
  - Initialize S = { s }, d(s) = 0.
  - Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v):\, u \in S} d(u) + \ell_e,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Representing Shortest Paths

- For each vertex u maintain the previous vertex from the source to that vertex, prev(u)
- Can reconstruct the shortest path to any vertex by walking backwards from the vertex to the source

```
Initialize_Single_Source(V)
     for v ∈ V
          dist(v) = ∞
          prev(v) = null
     dist(s) = 0
```


Dijksta's algorithm pseudo-code single source all shortest paths

For every vertex dist [v]← ∞ as distance; prev[v]= null;
Dist[source] = 0;
Q = V (Q will be the set of unfinished vertices)
$V_T \neq \varnothing$
while Q is not empty.
// Loop invariant: V_T = { v: know shortest path}
// each time through finds shortest path to vertex
// and adds edge to V_T

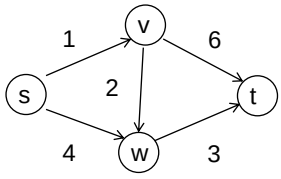
Dijksta's algorithm pseudo-code single source all shortest paths

- For every vertex dist [v]← ∞ as distance; prev[v]= null;
- Dist[source] = 0;
- Q = V (Q will be the set of unfinished vertices)
- $V_T \neq \varnothing$
- while Q is not empty. // Loop invariant: V_T = { v: know shortest path}
  // each time through finds shortest path to vertex and adds to V_T
  - u be vertex with smallest distance to source in Q, remove u* from Q
  - $V_T = V_T \cup \{u^*\}$
  - for all of its unfinished neighbors v
    - » calculate their tentative distance to source through current node --- u*.
    - » If this distance is less than the previously recorded tentative distance of v, then overwrite dist[v] and update prev[v]


```
Algorithm: Prim-MST (G)
Input: Graph G=(V,E) with edge-weights.
 // Initialize priorities and place in priority queue.
1.   priority[i] = infinity for each vertex i
2.   Insert vertices and priorities into
priorityQueue;
  // Set the priority of vertex 0 to 0.
3.   priorityQueue.decreaseKey (0, 0)  //

// Process vertices one by one in order of priority

4.   while priorityQueue.notEmpty()
        // Get "best" vertex out of queue.
5.      v = priorityQueue.extractMin()
6.      Add v to MST;
           // Explore edges from v.
7.      for each edge e=(v, u) in adjList[v]
8.         w = weight of edge e=(v, u);
// If it's shorter to get to MST via v, then update.
9.         if priority[u] > w
10.           priorityQueue.decreaseKey (u, w)
11.           predecessor[u] = v
12.        endif
13.     endfor
14.  endwhile

15.  Build MST;
16.  return MST
```

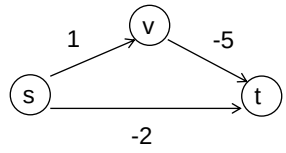
```
Algorithm: Dijkstra-SPT (G, s)
Input: Graph G=(V,E) with edge weights ≥ 0
// Initialize priorities & put in priority queue.
1. priority[i] = infinity for each vertex i;
2. Insert vertices and priorities into
priorityQueue;
   // Source s has priority 0
3. priorityQueue.decreaseKey (s, 0)

// Process vertices one by one in order of priority

4. while priorityQueue.notEmpty()
      // Get "best" vertex out of queue.
5.    v = priorityQueue.extractMin()
6.    Add v to SPT;
         // Explore edges from v.
7.    for each edge e=(v, u) in adjList[v]
8.       w = weight of edge e=(v, u);
// If it's shorter to get to u from s via v, update.
9.    if priority[u] > priority[v] + w
10.      priorityQueue.decreaseKey (u,
priority[v]+w)
11.      predecessor[u] = v
12.   endif
13.   endfor
14. endwhile

15. Build SPT;
16. return SPT
```

Simple Examples



Works!

What if negative edges lengths?
-- What if run Dijkstra's on this?
-- Add constant to each edge?



Dijksta's algorithm – proof of correctness

- Theorem: For every directed graph with non-negative edge lengths, Dijkstra's algorithm correctly computes all shortest path distances

Proof by induction:

Picture

Dijksta's algorithm – proof of correctness

Proof by induction: mark nodes with a * once shortest path has been found

base case: - empty path is optimal

Inductive case: all previous iterations correct → next iteration correct

1. Dijkstra picks next node to add to the shortest path tree w by finding the w with the smallest length path given by dist(v*) + l(v*, w) where it has already found the shortest path to v*
   ---- must show this is the shortest path of all possible paths from s to w*.

   Know the shortest path to v* has been found in a previous iteration by the inductive hypothesis
2. Suppose there is a shorter path, P_2, s to w that has not been found yet. Any path from s to w must cross the frontier between nodes whose shortest paths are known and other nodes. Let the edge where P_2 crosses the frontier be (y*, z) thus there is a path from s to y* followed by the edge (y*, z), then a path from z to w.
3. But the the last section of P_2 must be ≥0 (since all edge lengths are non-negative).
4. Finally consider the path length of the part of P_2 from s to z given by dist(s to y*)+ l(y*, z). But the path from s to z that is part of the proposed better path from s to W* must be ≤ shortest path from s to w. Why?. This contradicts that the vertex chosen by Dijkstra is the closest vertex to s among all the vertices one edge away from the vertices for which the shortest path is already known.

Copyright ©The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Kenneth H. Rosen, *Discrete Mathematics and its Applications*, 7e

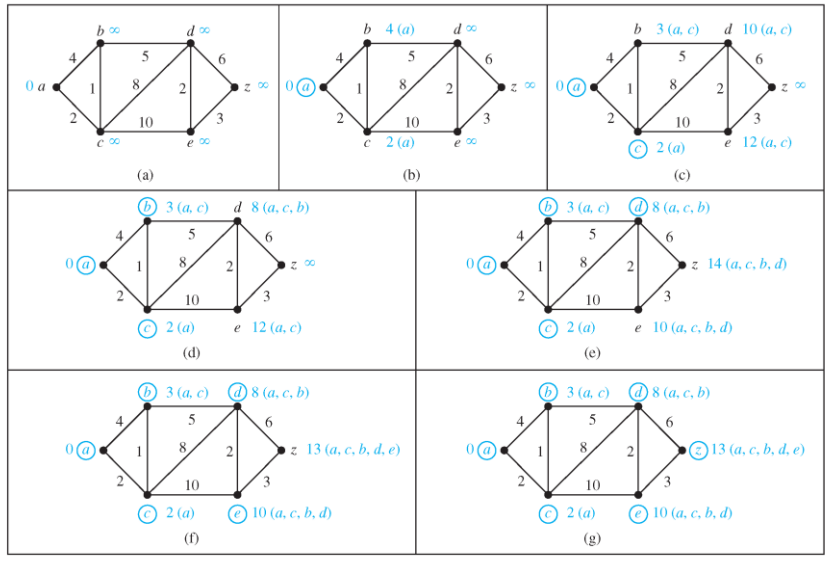


FIGURE 4 **Using Dijkstra's Algorithm to Find a Shortest Path from *a* to *z*.**

Notes on Dijkstra's algorithm

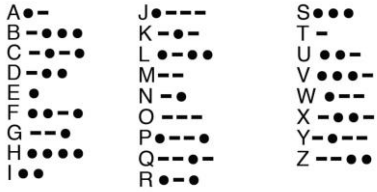
- Doesn't work for graphs with negative weight
- Applicable to both undirected and directed graphs
- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue
- Don't mix up Dijkstra's algorithm with Prim's algorithm!

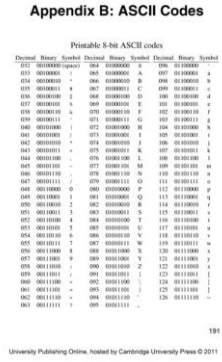
Coding Problem

- *Coding*: assignment of bit strings to alphabet characters
- *Codeword*: bit string assigned to character
- Two types of codes:
  - *fixed-length encoding* (e.g., ASCII)
  - *variable-length encoding* (e,g., Morse code)
- *Prefix-free codes*: no codeword is a prefix of another codeword

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

Morse and ASCII codes




Appendix B: ASCII Codes

Printable 8-bit ASCII codes

Decimal	Binary	Symbol	Decimal	Binary	Symbol	Decimal	Binary	Symbol
032	00100000	(space)	064	01000000	@	096	01100000	`
033	00100001	!	065	01000001	A	097	01100001	a
034	00100010	"	066	01000010	B	098	01100010	b
035	00100011	#	067	01000011	C	099	01100011	c
036	00100100	$	068	01000100	D	100	01100100	d
037	00100101	%	069	01000101	E	101	01100101	e
038	00100110	&	070	01000110	F	102	01100110	f
039	00100111	'	071	01000111	G	103	01100111	g
040	00101000	(	072	01001000	H	104	01101000	h
041	00101001	)	073	01001001	I	105	01101001	i
042	00101010	*	074	01001010	J	106	01101010	j
043	00101011	+	075	01001011	K	107	01101011	k
044	00101100	,	076	01001100	L	108	01101100	l
045	00101101	-	077	01001101	M	109	01101101	m
046	00101110	.	078	01001110	N	110	01101110	n
047	00101111	/	079	01001111	O	111	01101111	o
048	00110000	0	080	01010000	P	112	01110000	p
049	00110001	1	081	01010001	Q	113	01110001	q
050	00110010	2	082	01010010	R	114	01110010	r
051	00110011	3	083	01010011	S	115	01110011	s
052	00110100	4	084	01010100	T	116	01110100	t
053	00110101	5	085	01010101	U	117	01110101	u
054	00110110	6	086	01010110	V	118	01110110	v
055	00110111	7	087	01010111	W	119	01110111	w
056	00111000	8	088	01011000	X	120	01111000	x
057	00111001	9	089	01011001	Y	121	01111001	y
058	00111010	:	090	01011010	Z	122	01111010	z
059	00111011	;	091	01011011	[	123	01111011	{
060	00111100	<	092	01011100	\	124	01111100	|
061	00111101	=	093	01011101	]	125	01111101	}
062	00111110	>	094	01011110	^	126	01111110	~
063	00111111	?	095	01011111	_			

191

University Publishing Online, hosted by Cambridge University Press © 2011

Example

- Let Σ = { lower case letters, five punctuation marks and space}
- How can we encode this – 5 bits since $2^5 = 32$
- Is there a way to reduce the length not of the code for each symbol BUT the average length of a message.
- Use frequency of the occurrence of the symbols
  - e, t, a - most frequent -- q, j, x, z – least frequent
  - Normalize so the sum of frequencies is = 1
  - Use frequencies to compute E(symbol length)
  - E.g. Morse code
- Prefix free codes are codes: for all symbols x and y – codeword(x) is not a prefix of codeword(y)
- Decoding is easy - - why?

Huffman codes – key insights

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves
- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows

.

Example

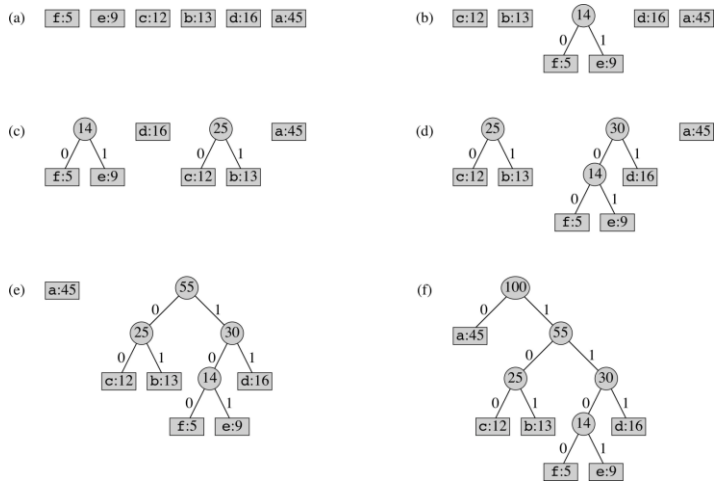
character	A	B	C	D	E
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

average bits per character: 2.25
for fixed-length encoding: 3
compression ratio: (3-2.25)/3*100% = 25%

Huffman's algorithm

- Initialize *n* one-node trees with alphabet characters and the tree weights with their frequencies.
- Repeat the following step *n*-1 times:
  - join two binary trees with smallest weights into one (as left and right subtrees)
  - make the new tree weight equal the sum of the weights of the two subtrees.
- Mark edges leading to left and right subtrees with 0's and 1's, respectively

Constructing a Huffman code tree



Example: Build tree (smallest to left = 0)

character	A	B	C	D	E
frequency	0.32	0.25	0.2	0.18	0.05

Codewords:

H.C. average bits per character:

Fixed-length bits per character:

compression ratio =

Decode: 011110111011011

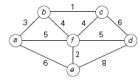
Huffman Code Proof Optimality: Big Picture

- Show that optimal prefix free code corresponds to a full binary tree.
- Huffman's algorithm puts two lowest frequency symbols as siblings of a full binary tree
- There exists an optimal code where the two symbols are at the lowest level of the tree, call these y and z
- Combining the two sibling symbols, y and z into w ABL(T') + f_y + f_z = ABL(T) for any full binary tree
- Proof by induction: base case – 2 symbols– only one tree inductive case: true for k-1 symbols → true for k symbols
  - » Assume ABL (T_H) > ABL (T_{Opt}) → ABL (T'_H) > ABL (T'_{Opt}) gives contradiction using the above identity

Extensions and issues

- Image compression
  - Fraction of a bit for a white pixel, higher for black pixel
  - Video/audio – only send changes
- Adaptive encoding
- Many schemes are more effective for particular applications

Supplemental



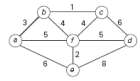
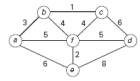
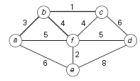
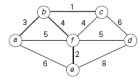
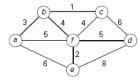
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 9.3 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

Developing and remembering Prim's Algorithm

Big idea: grow a tree adding one vertex (closest) at a time -- Refinements

1. How find next vertex? -- what needs to happen to set up for finding the next vertex?
2. What are the operations needed?
3. What are possible data structures?
4. What should control flow look like?

Developing and remembering Prim's Algorithm - 2

Try to make it concrete

- Keep track of vertices in a list and associate with each the "edge connecting it to the tree and the edge weight"
- Store the graph either as an adjacency matrix or an adjacency list
- Initialize the list of vertices, mark the start vertex with nil, 0 since it has no edge and no weight
- Traverse the adjacency list of the start vertex and update distances and edges
- Loop till all the vertices are connected – n-1
  - remove the nearest vertex and mark it in the tree
  - update the distances and edges of vertices in the adjacency list

Single Source Shortest Paths Problem Dijksta's algorithm

- Doesn't BFS solve shortest paths?
  Yes, but only if the edge costs are all 1
- Why not just add edges to get the weight? (Good Question, can you use what you already know.)

Dijksta's algorithm pseudo-code single source all shortest paths

- Doesn't BFS solve shortest paths?
  Yes, but only if the edge costs are all 1
- Why not just add edges to get the weight? (Good Question, can you use what you already know.)

- But problem is you may increase the number of edges significantly

Dijkstra's Algorithm: Implementation

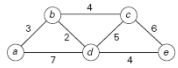
- For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e$.
  - Next node to explore = node with minimum $\pi(v)$.
  - When exploring v, for each incident edge e = (v, w), update

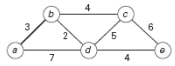
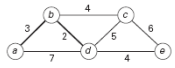
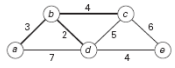
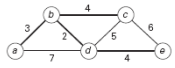
$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

- Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

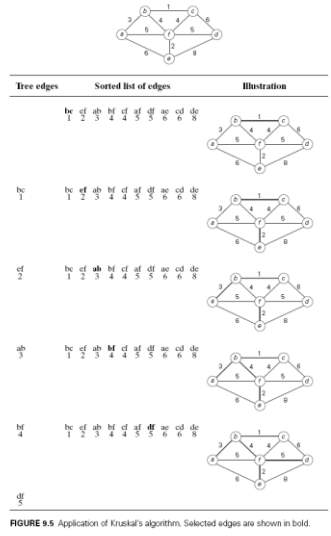
The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b:	$a - b$	of length 3
from a to d:	$a - b - d$	of length 5
from a to c:	$a - b - c$	of length 7
from a to e:	$a - b - d - e$	of length 9

FIGURE 9.11 Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

Another greedy algorithm for MST: Kruskal's

- Sort the edges in non-decreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests F_1, F_2, …, F_{n-1}
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

FIGURE 9.5 Application of Kruskal's algorithm. Selected edges are shown in bold.

Notes about Kruskal's algorithm

- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)
- Cycle checking: a cycle is created if and only if added edge connects vertices in the same connected component
- *Union-find* algorithms – see section 9.2

Correctness and Analysis: Kruskal's Algorithm

- Correctness : apply the cut property at each stage of the algorithm
- Analysis: Assuming n vertices and m edges. Using Union, Find and MakeSet operations discussed in the text with an efficient implementation
  - the algorithms order of complexity is dominated by the sorting operation on the edges hence it is O(|E| * (log |E|)

Bellman Ford Algorithm

```
Bellman-Ford (G, w, s)// w: weights
   Initialize_Single_Source(V)
   For i = 1 to |V|-1
         for each edge (u,v) in E
            Relax((u,v),w)
   for each edge (u,v) in E
      if dist(v) > dist(u) + w(u,v)
         return false   //negative cycle detected
```

Correctness of Bellman Ford Algorithm

- May want to defer till get to dynamic programming. Good lab to write out as a dynamic programming algorithm
- Proof of correctness is proof of optimal substructure

DAG Algorithm

- DAG implies no cycles – no need to worry about negative weight cycles
- If we topologically sort the vertices, then any path must be consistent with the topological sort, that is the u comes before v in the path $\leftrightarrow$ u comes before v in the topological sort
- By making one pass over the vertices in the topological sort and relaxing the values of the vertices in their adjacency list we are guaranteed to find the shortest paths.

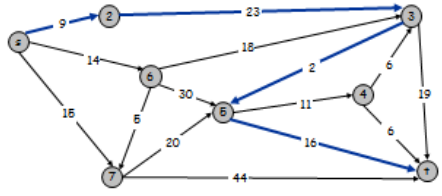
DAG algorithm pseudo code and efficiency

```
DAG-Shortest-Paths (G,w,s)
   Topologically sort vertices in G
   Initialize-Single-Source (G,s)
   For each vertex u // in topological sorted order
      For vertex v in the adjacency list of u
          Relax(u,v,w)
```

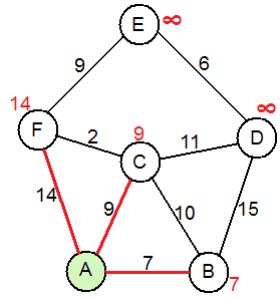
- Running time is $\theta(|E|+|V|)$ since topological sorting is $\theta(|E|+|V|)$ and the Relax function is called $\theta(|E|+|V|)$ times

Shortest Path Problem

- Shortest path network.
  - Directed graph G = (V, E).
  - Source s, destination t.
  - Length ℓ_e = length of edge e.
  - cost of path = sum of edge costs in path
- Shortest path problem: find shortest directed path from s to t.



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.



A* (heuristic): Brief comparison

- A* uses finds a least-cost path from a given initial node to one goal node (out of one or more possible goals).
- It uses a distance-plus-cost heuristic function f(x) = g(x) + h(x)) to determine the order in which the search visits nodes in the tree.
  - g(x) – path cost function which is the cost from the starting node to the current node
  - h(x) an ***admissible*** "heuristic estimate" of the distance to the goal (usually denoted). It must not over-estimate the distance to the goal